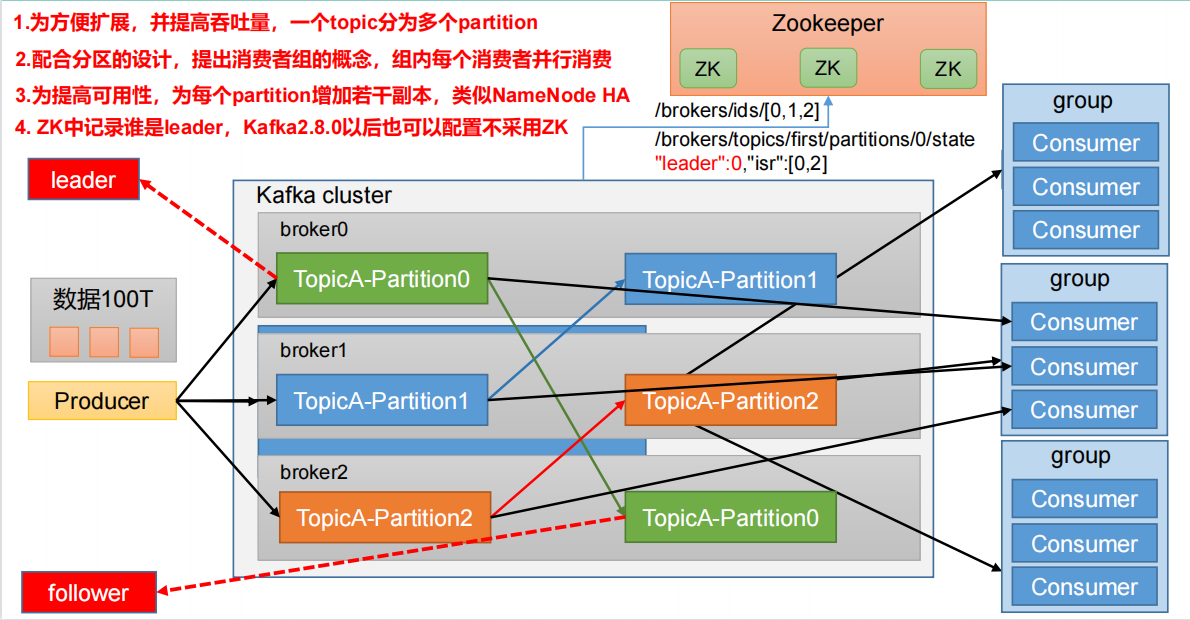
# Kafka总结

# 什么是kafka以及kafka的基础架构

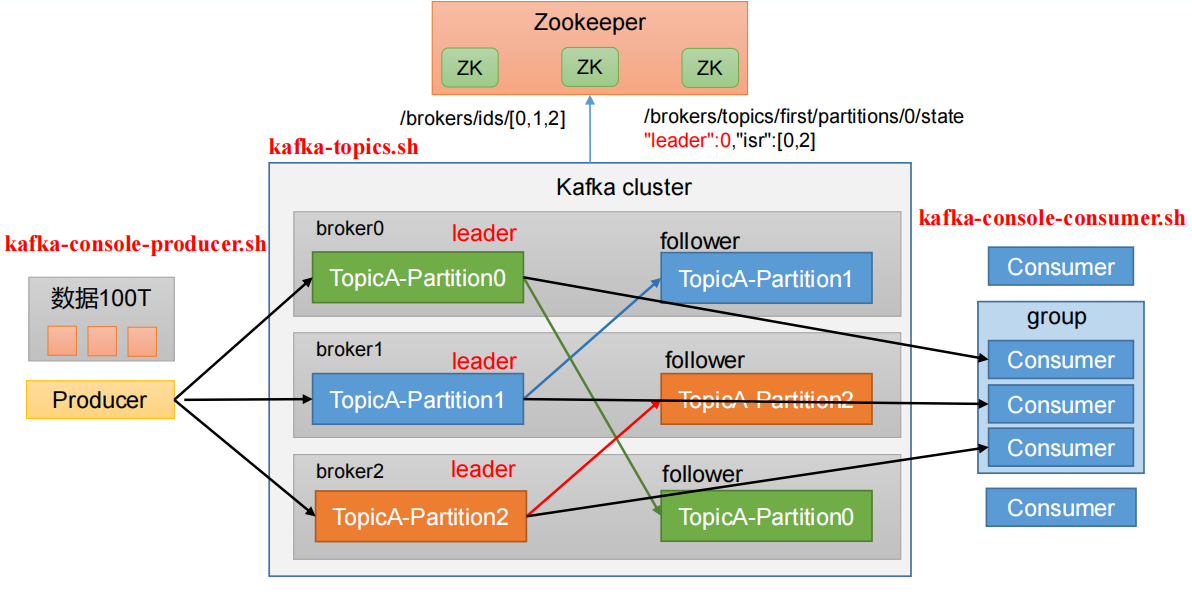
Kafka是一个高吞吐的分布式的消息系统，是基于发布/订阅模式的消息队列。



## 术语

* Producer：消息生产者，就是向 Kafka broker 发消息的客户端。
* Consumer：消息消费者，向 Kafka broker 拉取消息的客户端。
* Consumer Group（CG）：消费者组，由多个 consumer 组成。消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。
* Broker：一台 Kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个broker 可以容纳多个 topic。
* Topic：可以理解为一个队列，生产者和消费者面向的都是一个 topic。
* Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，每个broker上的topic叫做一个partition（一个 topic 可以分为多个 partition），每个 partition 是一个有序的队列。
* Replica：副本。一个 topic 的每个分区都有若干个副本，即一个 Leader 和若干个Follower。
* Leader：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 Leader。
* Follower：每个分区多个副本中的“从”，实时从 Leader 中同步数据，保持和Leader 数据的同步。Leader 发生故障时，某个 Follower 会成为新的 Leader。
* ISR: in-sync replica set（ISR）, 意为和Leader保持同步的Follower+Leader集合(leader：0，isr:0,1,2)。如果Follower长时间未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR。该时间阈值由replica.lag.time.max.ms参数设定，默认30s。例如2超时，(leader:0, isr:0,1)。
* AR: Kafka分区中的所有副本统称。
* OSR ：表示 Follower 与 Leader副本同步时，延迟过多的副本。AR=ISR+OSR
* LEO（Log End Offset）：每个副本的最后一个offset，LEO其实就是最新的offset + 1。
* HW（High Watermark）：所有副本中最小的LEO 。

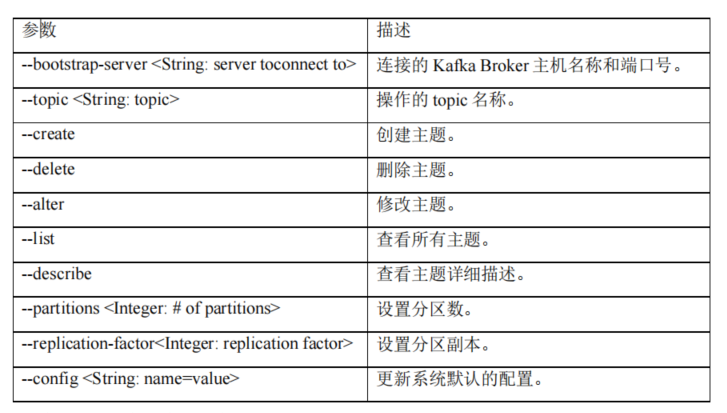
# Kafka 命令行操作



## 3.1、主题命令行操作

1）查看操作主题命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh



2）查看当前服务器中的所有 topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 –list

3）创建 first topic

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --create --partitions 1 --replication-factor 3 --topic first

选项说明：

--topic 定义 topic 名

--replication-factor 定义副本数

--partitions 定义分区数

4）查看 first 主题的详情

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --describe --topic first

5）修改分区数（注意：分区数只能增加，不能减少）

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --alter --topic first --partitions 3

6）再次查看 first 主题的详情

[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --describe --topic first

7）删除 topic（学生自己演示）

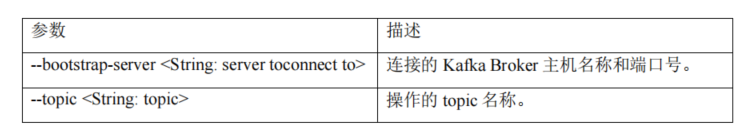
[atguigu@hadoop102 kafka]$ bin/kafka-topics.sh --bootstrap-server

hadoop102:9092 --delete --topic first

## 3.2、生产者命令行操作

1）查看操作生产者命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh



2）发送消息

[atguigu@hadoop102 kafka]$ bin/kafka-console-producer.sh --

bootstrap-server hadoop102:9092 --topic first

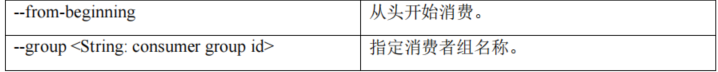
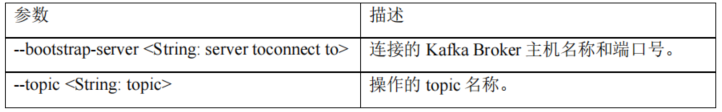
>hello world

>atguigu atguigu

## 3.3、消费者命令行操作

1）查看操作消费者命令参数

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh



2）消费消息

（1）消费 first 主题中的数据。

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --

bootstrap-server hadoop102:9092 --topic first

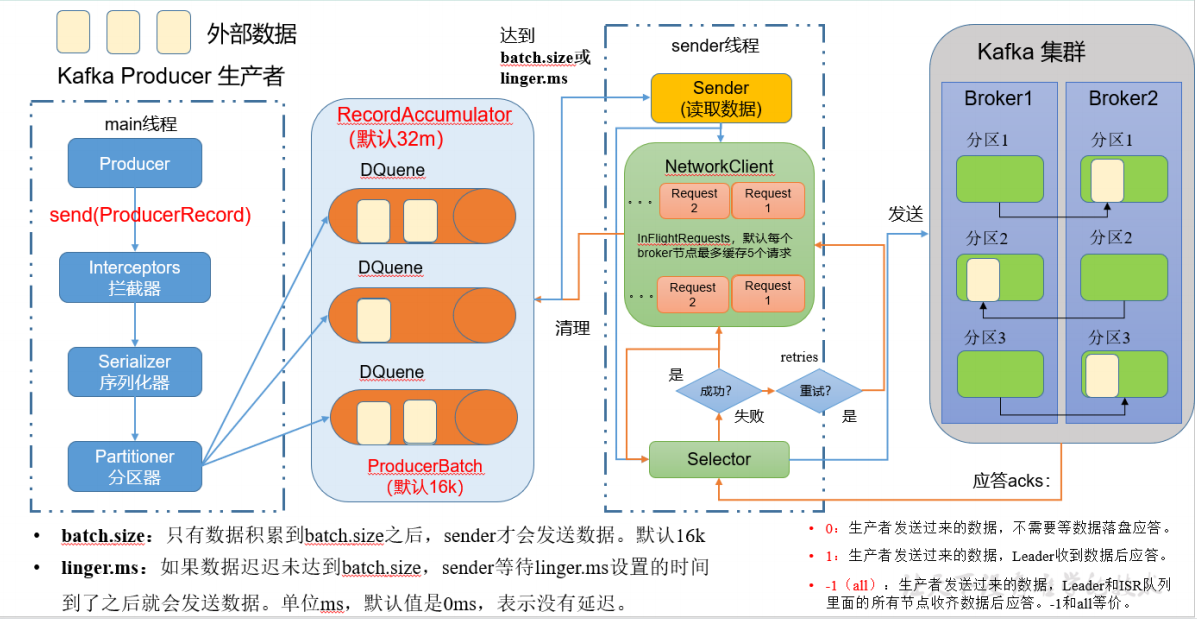
（2）把主题中所有的数据都读取出来（包括历史数据）。

[atguigu@hadoop102 kafka]$ bin/kafka-console-consumer.sh --

bootstrap-server hadoop102:9092 --from-beginning --topic first

# Kafka 生产者

## 4.1、生产者消息发送流程



**外部数据 ----> 存储到kafka集群中**

在数据发送的过程中会涉及到两个线程——main 线程和 Sender 线程，当接受外部传过来的数据的时候，会先创建一个main线程，在main线程中创建producer对象，然后调用send方法，将数据进行发送，然后会经过拦截器，对发送的数据进行处理、加工，再经过序列化器，对传输的数据进行序列化，在根据分区器的分区策略对传输的数据进行分区处理，到达缓冲区

（RecordAccumulator记录收集器，里面维护了一个一个双端的缓存队列和一个内存池，内存池的默认大小是32m，当向双端队列中发送数据的时候，会创建批次大小，当创建批次的时候，会从内存池中取出内存，当数据发送到kafka集中后，清理批次的时候，会释放内存，放回到内存池中）中指定的分区（一个分区会创建一个队列），在分区中按照批次的方式进行存储数据，每一个批次的默认大小（batch.size）为16k，当一个批次的（数据大小积累到batch.size（默认16k）大小时）或者（数据没有积累到batch.size，但是到达了延迟时间（linger.ms 默认是0ms，表示没有延迟）），唤醒sender线程，然后会发送请求从分区中拉取数据，拉取数据的方式是以brokerId为key,所有分区的请求为value放到队列中，如果发送数据的第一个请求到达集群中的某一个broker没有应答，允许继续发送请求，默认每个broker节点最多缓存5个请求，通过selector发送数据，数据发送成功之后，会有应答机制，返回acks，应答级别有3种，如果反馈回来的请求是成功，则会删除发送数据成功的请求以及清理分区中请求中拉取的数据，如果失败会进行重试，重试的次数（默认是Int的最大值，可以进行修改，一般是3-5次），

0：生产者发送过来的数据，不需要等数据落盘应答。1：生产者发送过来的数据，Leader收到数据后应答。 -1（all）：生产者发送过来的数据，Leader和ISR队列，里面的所有节点收齐数据后应答。-1和all等价。

## 4.2、异步发送 API

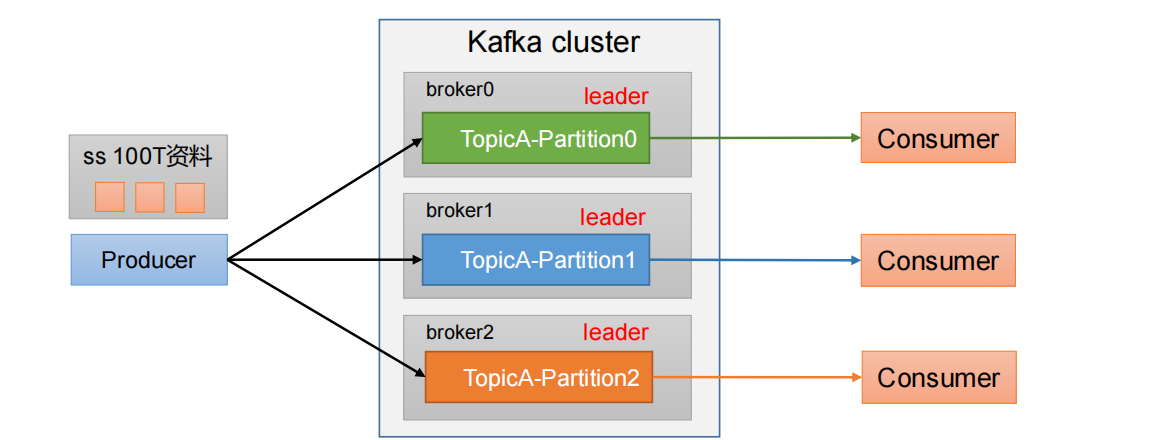
## 4.3、同步发送 API

## 4.4、生产者分区

### 4.4.1、分区的好处

（1）便于合理使用存储资源，每个Partition在一个Broker上存储，可以把海量的数据按照分区切割成一块一块的数据存储在多台Broker上。合理控制分区的任务，可以实现负载均衡的效果。

（2）提高并行度，生产者可以以分区为单位发送数据；消费者可以以分区为单位进行消费数据。



### 4.4.2、生产者发送消息的分区策略

（1）指明partition的情况下，直接将指明的值作为partition值；

例如partition=0，所有数据写入分区0

（2）没有指明partition值，但是有key（一般是数据库表的表名字）的情况下，将key的hash值与topic的partition数进行取余得到partition值；

例如：key1的hash值=5， key2的hash值=6 ，topic的partition数=2，那 么key1 对应的value1写入1号分区，key2对应的value2写入0号分区。

（3）既没有partition值又没有key值的情况下，Kafka采用Sticky Partition（黏性分区器），会随机选择一个分区，并尽可能一直使用该分区，待该分区的当前batch已满或者当前批次的等待时间到了，Kafka再随机一个分区进行使用（和上一次的分区不同）。

例如：第一次随机选择0号分区，等0号分区当前批次满了（默认16k）或者linger.ms设置的时间到，Kafka再随机一个分区进行使用（如果还是0会继续随机）。

### 4.4.3、自定义分区

#### 4.4.3.1、实现步骤

（1）自定义类实现 Partitioner 接口。

（2）重写 partition()方法。

（3）使用分区器中的方法，在生产者的配置中添加分区器参数。

|  |
| --- |
| package com.song.kafka\_demo.producer;  import org.apache.kafka.clients.producer.Partitioner;  import org.apache.kafka.common.Cluster;  import java.util.Map;  public class MyPartitioner implements Partitioner {  @Override  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {  // 获取数据 song  String msgValues = value.toString();  int partition;  if (msgValues.contains("song")) {  partition = 0;  } else {  partition = 1;  }  return partition;  }  @Override  public void close() {  }  @Override  public void configure(Map<String, ?> configs) {  }  }  ===============================================  // 关联自定义分区器  properties.put(ProducerConfig.PARTITIONER\_CLASS\_CONFIG,"com.song.kafka\_demo.producer.MyPartitioner"); |

## 4.5、生产者如何提高吞吐量

• RecordAccumulator：修改缓冲区大小，默认32m，修改为64m

• batch.size： 修改批次大小，默认16k

• linger.ms：修改等待时间，修改为5-100ms一次拉一个，来了就走

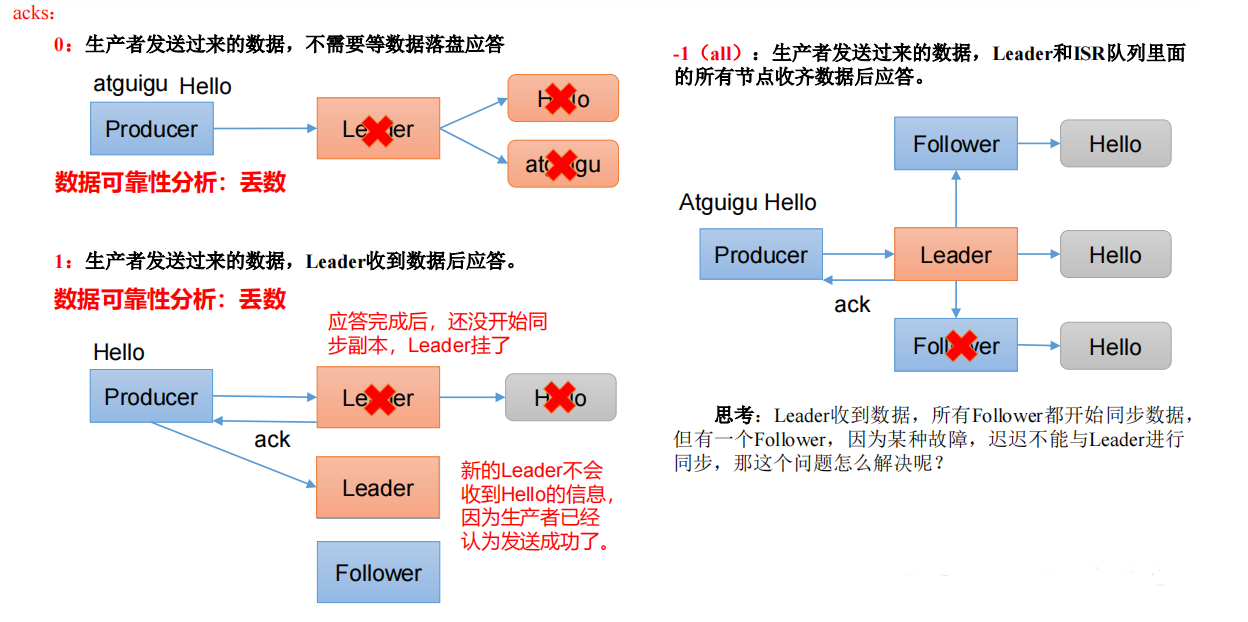
• compression.type：压缩snappy

## 4.6、数据可靠性

### 4.6.1、ack 应答原理

#### 4.6.1.1、应答级别

* 0：生产者发送过来的数据，不需要等数据落盘应答。
* 1：生产者发送过来的数据，Leader收到数据后应答。
* -1（all）：生产者发送过来的数据，Leader和ISR队列，里面的所有节点收齐数据后应答。-1和all等价。

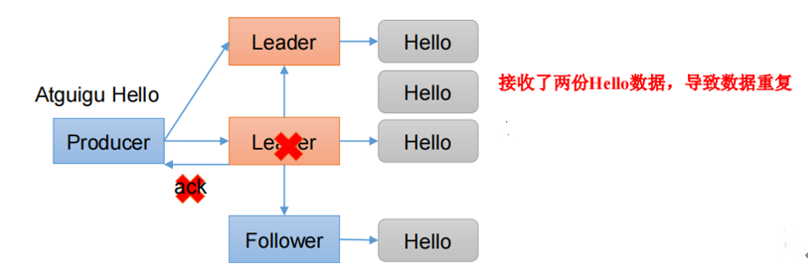


1. Acks为0，生产者发送过来的数据，不需要等数据落盘应答，但是有可能在落盘的过程中，leader服务器挂了，但是已经返回成功的信息，所以则会导致丢失数据。
2. Acks为1，生产者发送过来的数据，Leader收到数据后再去应答，但是有可能在应答完成之后，还没有开始同步副本数据，Leader挂了，然后会在剩余的follower中重新选择出一个Leader，但是新的Leader不会接受到之前发送的数据，因为之前的Leader已经返回发送成功的消息了，所以则会导致数据丢失。
3. Acks为-1，生产者发送过来的数据，Leader和ISR队列中的所有节点都收齐数据之后才去应答，但是如果其中的一个Follower出现故障迟迟不能应答怎么办？在Leader中维护了一个动态的ISR（in-sync replica set），意为和Leader保持同步的Follower+Leader集合(leader：0，isr:0,1,2（brokerId）)，如果在30s（该时间阈值由replica.lag.time.max.ms参数设定，默认30s）内，Follower未向Leader发送通信请求或同步数据，则该Follower将被踢出ISR。

#### 4.6.1.2、最终解决方式

数据完全可靠条件 = ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2，

**但是使用此方式会造成数据重复性问题，解决方式看下面的数据去重**



acks： -1（all）：生产者发送过来的数据，Leader和ISR队列里面的所有节点收齐数据后应答，接受完数据之后，在未进行应答反馈之前，leader挂掉，会在follower中重新选举leader，producer未接受到成功的反馈，所以回向leader再次发送数据，就造成了消息重复的问题。

### 4.6.2、可靠性总结

* acks=0，生产者发送过来数据就不管了，可靠性差，效率高；
* acks=1，生产者发送过来数据Leader应答，可靠性中等，效率中等；
* acks=-1，生产者发送过来数据Leader和ISR队列里面所有Follwer应答，可靠性高，效率低；

**在生产环境中：**

* acks=0很少使用；
* acks=1，一般用于传输普通日志，允许丢个别数据；
* acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景。

## 4.7、数据去重

### 数据传递语义

* 至少一次（At Least Once）= ACK级别设置为-1 + 分区副本大于等于2 + ISR里应答的最小副本数量大于等于2
* 最多一次（At Most Once）= ACK级别设置为0
* 精确一次（Exactly Once）：对于一些非常重要的信息，比如和钱相关的数据，要求数据既不能重复也不丢失。

**总结：**

* At Least Once可以保证数据不丢失，但是不能保证数据不重复；
* At Most Once可以保证数据不重复，但是不能保证数据不丢失。

### 精确一次

 精确一次（Exactly Once）：对于一些非常重要的信息，比如和钱相关的数据，要求数据既不能重复也不丢失。依赖于幂等性和事务

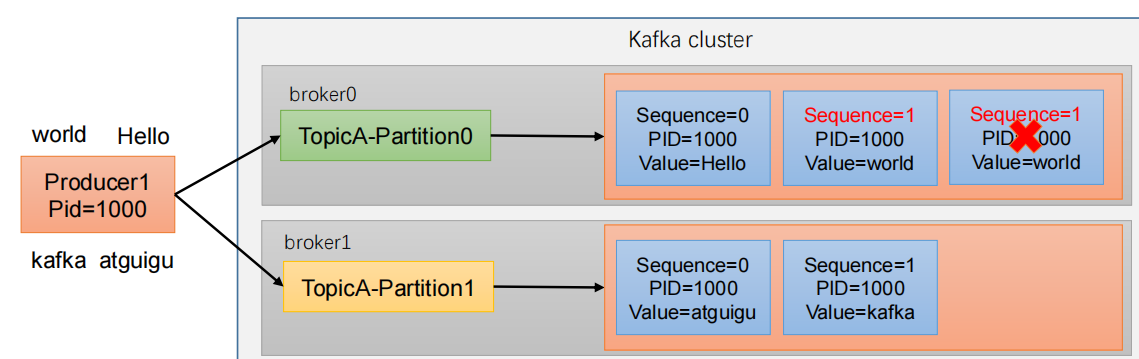
#### 4.7.2.1、幂等性

幂等性就是指Producer不论向Broker发送多少次重复数据，Broker端都只会持久化一条，保证了不重复。

精确一次（Exactly Once） = 幂等性 + 至少一次（ ack=-1 + 分区副本数>=2 + ISR最小副本数量>=2）。

重复数据的判断标准：具有<PID, Partition, SeqNumber>相同主键的消息提交时，Broker只会持久化一条。其中PID是Kafka每次重启都会分配一个新的；Partition 表示分区号；Sequence Number是单调自增的。

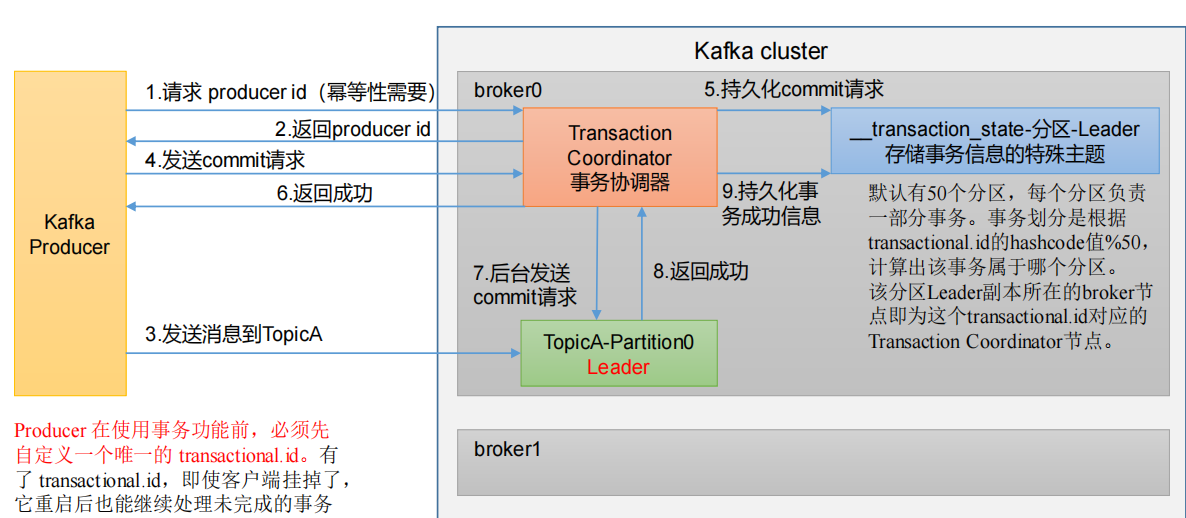
所以幂等性只能保证的是在单分区单会话内不重复。



开启幂等性参数 enable.idempotence 默认为 true，false 关闭幂等性。如果想要在整个集群中保证数据不重复，则还需要依赖于事务。

#### 4.7.2.2、事务

开启事务的前提是必须开启幂等性。



事务的实现主要是依赖于事务协调器（Transaction Coordinator），但是在kafka中每一个broker都有事务协调器，所以应该使用哪一个事务协调器？

首先Producer 在使用事务功能前，必须先自定义一个唯一的 transactional.id，事务id不但可以判断出要使用哪一个事务协调器，还能够保证在在客户端挂掉，重新启动之后也能继续处理未完成的事务。

在每一个broker中都存在一个存储事务信息的特殊主题(\_\_transaction\_state-分区-Leader)，在这个Topic中默认有50个分区，每个分区负责一部分事务，事务划分是根据设置的transactional-id的哈希值计算对\_transaction\_state分区数（默认值是50）取余运算，找到分区编号，该分区对应的leader副本所对应的broker的即为Tranaction Coordinator所在节点。

（1）Producer向任意一个brokers发送获取事务协调器的请求（FindCoordinatorRequest）来获取Transaction Coordinator的地址。

（2）producer向事务协调器发起获取PID（幂等性需要）的请求，返回PID

（3）然后producer发送消息到topic分区中,并且同时会向事务协调器发起持久化commit请求，事务协调器接收到持久化commit请求后，会将这个请求保存到存放事务的topic中去。

（4）接着事务协调器会向生产者返回成功的信息，并且会向topic分区中发送commit请求，判断topic分区中的消息是否持久化成功，如果处理完毕，则向事务协调器返回成功信息，最后事务协调器会向存储事务的topic主题中进行事务成功的记录。

## 4.8、生产经验——数据有序

## 4.9、生产经验——数据乱序

# Kafka Broker

## 5.1、Kafka Broker 工作流程

### 5.1.1、Zookeeper 存储的 Kafka 信息

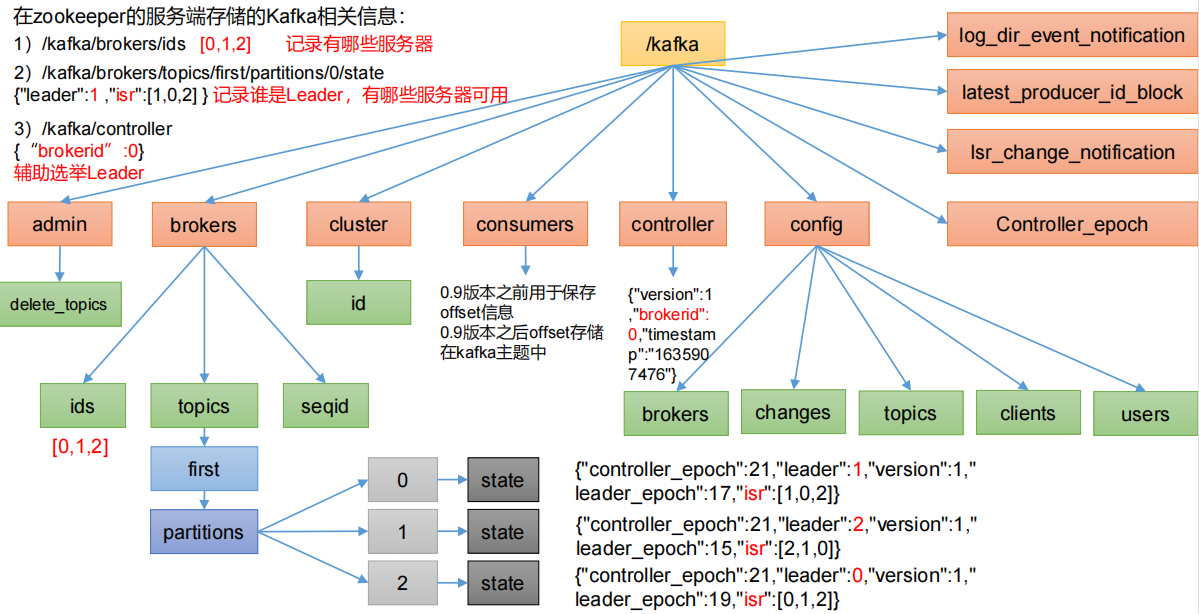
（1）启动 Zookeeper 客户端。

[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkCli.sh

（2）通过 ls 命令可以查看 kafka 相关信息

[zk: localhost:2181(CONNECTED) 2] ls /kafka

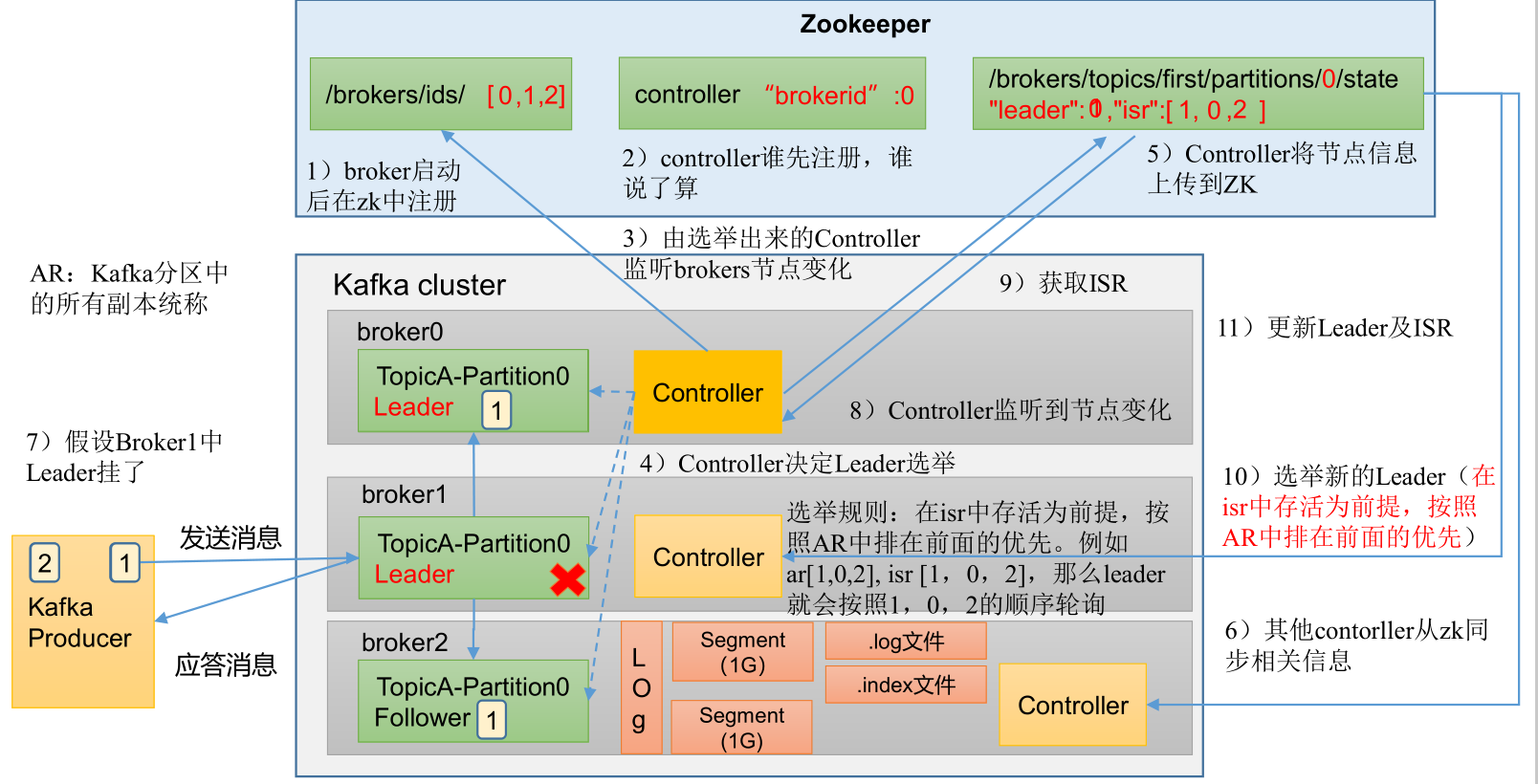
### 5.1.2、服务端存储的相关信息



主要有以下几个比较主要的

* Brokers
* Ids:主要存储的是brokerId
* Topics:记录的是所有的主题中的Leader信息以及ISR信息， topic(主题) --- partitions（分区）--- 分区号 --- state ---> 具体的Leader信息和ISR信息。
* Consumers
* 0.9版本之前用于保存offset信息，0.9版本之后offset存放的位置在kafka的主题中（topic），放到topic的原因，是因为每一个消费者消费信息都会更新offset，造成了kafka与zk进行频繁通信，效率较慢。
* Controller
* 辅助选举leader，在broker节点上都有一个controller模块，broker节点上的controller，才是真正决定谁是Leader。每个节点上都有Controller，那个节点说了算？这时候zookeeper上的Controller模块的用途就展现出来了，所有broker节点上的controller都会去抢先去注册zookeeper上的Controller，谁先抢占到注册权，谁才是负责Leader的选举的controller。

### 5.1.3、broker的工作流程



Broker启动之后，都会在zookeeper中进行注册，然后每一个broker节点上的controller就会去抢占zookeeper中的controller节点，谁抢先注册到controller节点，谁才是负责分区副本Leader的选举的controller，当某一个controller抢占到注册权后，会先监听zookeeper节点下的brokers(下面的ids)里面的节点变化（当节点发生变化的时候，会第一时间知道），然后再进行Leader的真正选举，Leader的选举策略首先是以在ISR中存活为前提，按照AR（所有的副本分区的统称）中排列在前面的顺序进行优先选举。例如ar[1,0,2], isr [1，0，2]，那么leader就会按照1，0，2的顺序轮询，AR的顺序是固定的，应该是在注册的时候确定的，选举出来topic分区副本的leader之后，会将Leader信息保存到zookeeper中topics节点下具体的topic节点中，其他broker节点中的controller会从zookeeper中同步相关的leader信息，其他的controller同步leader数据的目的是当已经抢占到注册权的controller宕机之后，能够更快的替代原先的controller，当选举出来的副本leader宕机之后，controller监控到brokers的节点的变化，会重新从zookeeper中拉取回leader信息和ISR信息，根据选举策略重新进行选举，选举完新的leader之后，会将新的Leader信息以及ISR信息更新到zookeeper中的topics节点下的具体topic节点中，其他的controller也会再次重新同步相关的数据。

### 5.1.4、服役新节点/退役旧节点

* 创建一个要均衡的主题。
* 生成一个负载均衡的计划。
* 创建副本存储计划。
* 执行副本存储计划。
* 验证副本存储计划。

## 5.3、Kafka 副本

### 5.3.1、副本基本信息

（1）Kafka 副本作用：提高数据可靠性。

（2）Kafka 默认副本 1 个，生产环境一般配置为 2 个，保证数据可靠性；太多副本会增加磁盘存储空间，增加网络上数据传输，降低效率。

（3）Kafka 中副本分为：Leader 和 Follower。Kafka 生产者只会把数据发往 Leader，然后 Follower 找 Leader 进行同步数据。

（4）Kafka 分区中的所有副本统称为 AR（Assigned Repllicas）。AR = ISR + OSR

ISR，表示和 Leader 保持同步的 Follower 集合。如果 Follower 长时间未向 Leader 发送通信请求或同步数据，则该 Follower 将被踢出 ISR。该时间阈值由 replica.lag.time.max.ms参数设定，默认 30s。Leader 发生故障之后，就会从 ISR中选举新的 Leader。OSR ，表示 Follower 与 Leader副本同步时，延迟过多的副本。

### 5.3.2、Leader及Follower数据同步问题

#### 5.3.2.1、Follower出现故障



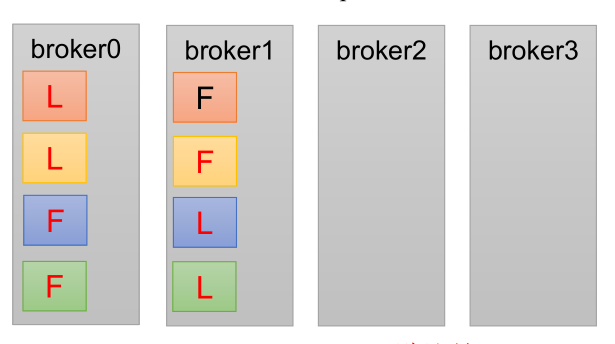
当Follower发生故障后会被临时踢出ISR，这个期间没有出现故障的Leader和Follower继续接收数据，等待该故障的Follower恢复正常之后，Follower会读取本地磁盘记录的上次的HW（高水位线），并且将Log文件中高于HW的数据给截掉，从HW的位置开始向Leader进行数据同步，等到该Follower 的LEO （副本的最后一个offset+1）大于等于该Partition 的HW的时候，即Follower追上Leader之后，就可以重新加入ISR了。

#### 5.3.2.2、Leader出现故障



当Leader发生故障之后，会从ISR中选出一个新的Leader，选出新的Leader后，会与其他分区进行比较，获取当前分区的HW，如果Follower中的数据高于Leader中的数据，为保证多个副本之间的数据一致性，所有的Follower会将各自的log文件中高于HW的部分数据进行截掉，然后从新的Leader中进行数据同步。

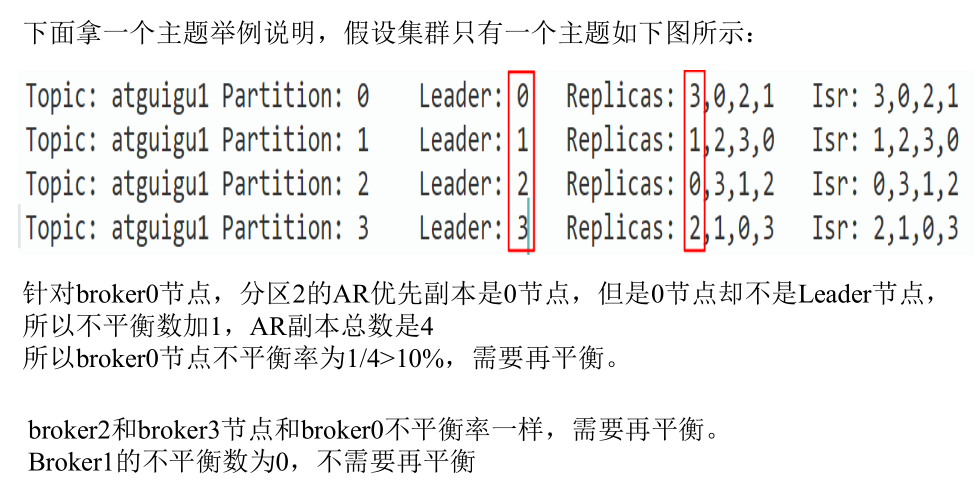
### 5.5.3、Leader Partition自动平衡



正常情况下，Kafka本身会自动把Leader Partition均匀分散在各个机器上，来保证每台机器的读写吞吐量都是均匀的。但是如果某些broker宕机，会导致Leader Partition过于集中在其他少部分几台broker上，这会导致少数几台broker的读写请求压力过高，其他宕机的broker重启之后都是follower partition，读写请求很低，造成集群负载不均衡。

解决方案，使用leader的再平衡，默认情况下再平衡机制是打开的（auto.leader.rebalance.enable，默认是true），每个broker允许的不平衡的leader的比率默认是10%，如果某个broker超过了这个值，控制器会触发leader的平衡。Kafka中检查leader负载是否平衡的间隔时间为5分钟（默认值300秒）。

计算不平衡的比例方式：

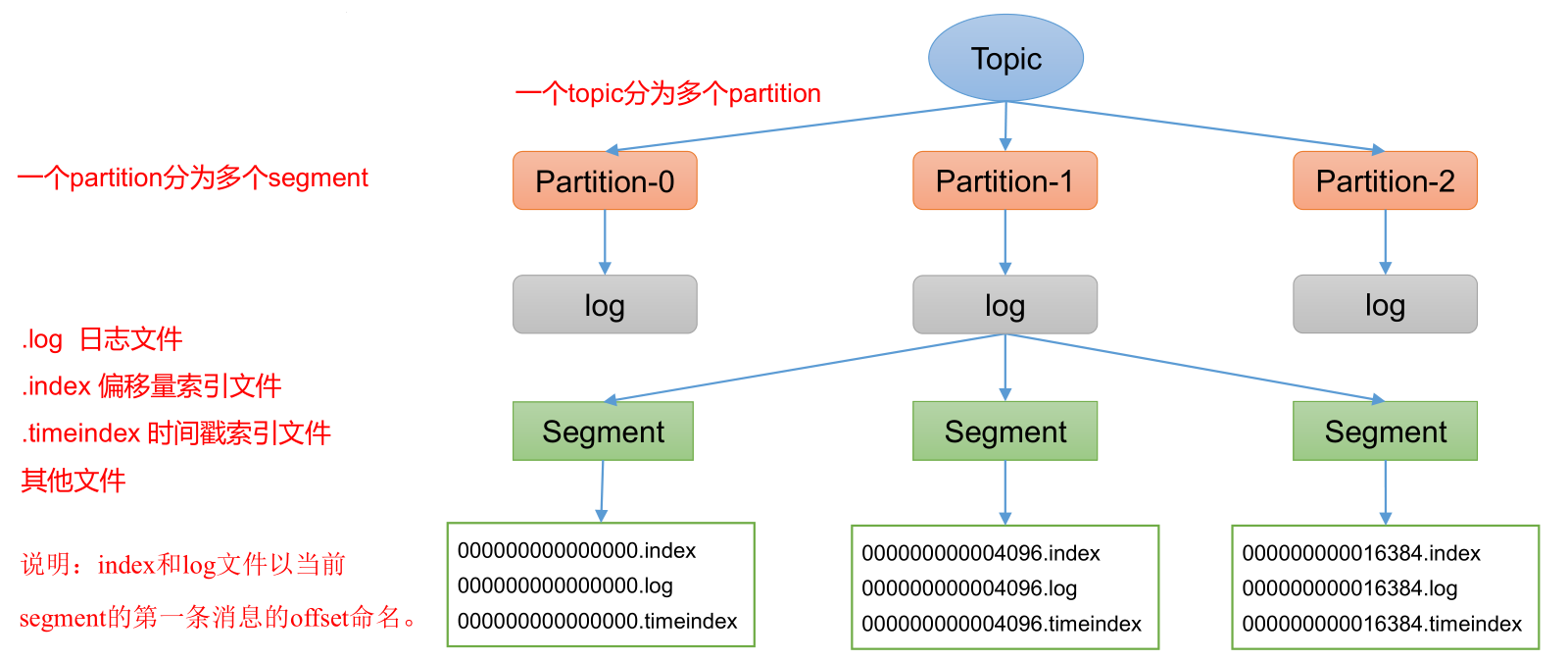


分区2的AR优先副本是0节点，但是0节点却不是Leader节点，说明发生了宕机。

但是一般情况下再平衡不建议开启，或者选择开启的话，再平衡比例可以选择大一些，因为触发再平衡机制后，在整个平衡的过程中是不能够处理数据的，所以再平衡会影响效率。

## 5.4、文件存储

### 5.4.1、Topic 数据的存储机制

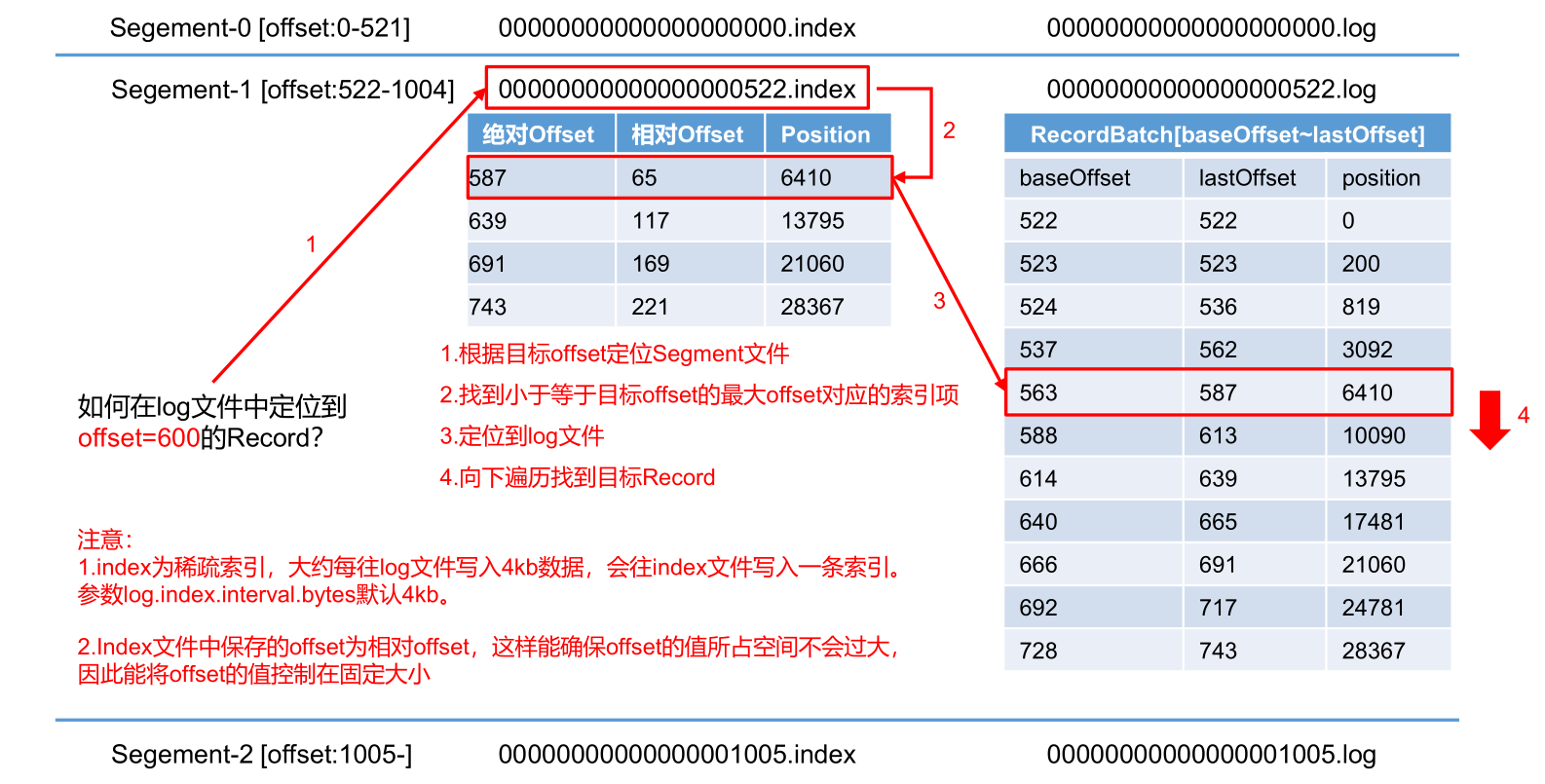


Topic是逻辑上的概念，而partition是物理上的概念，每个partition对应于一个log文件，该log文件中存储的就是Producer生产的数据。

Producer生产的数据会被不断追加到该log文件末端，为防止log文件过大导致数据定位效率低下，Kafka采取了分片和索引机制，将每个partition分为多个segment。每个segment包括：“.index”文件、“.log”文件和.timeindex等文件。

“.log”文件存放的是数据，“.index”存放的时候偏移量索引文件，“.timeindex”文件存放的是时间戳索引文件，而且“.log”和“.index”文件的命名规则是以当前segment的第一条消息的offset索引进行命名的，每个segment的默认大小是1G，这些文件位于一个文件夹下，该文件夹的命名规则为：topic名称+分区序号，例如：/opt/mode/kafka/data/first-0。

### 5.4.2、Log文件和Index文件详解



* .index为稀疏索引，大约每往log文件写入4kb数据，会往index文件写入一条索引。参数log.index.interval.bytes默认4kb。
* .Index文件中保存的offset为相对offset，这样能确保offset的值所占空间不会过大，因此能将offset的值控制在固定大小

**如何在log文件中定位到offset=600的Record？**

首先会根据offset去比较索引文件以及“.log”文件的文件名，定位到是哪一个segment文件，然后根据给定的offset，与（segment的起始offset+相对offset）然后进行比较，找到小于等于目标offset的最大offset对应的索引项的position的值，根据找到的position的值，去该segment下的“.log”文件中找对应的position位置，然后向下遍历寻找，就能找到目标offset=600的记录数据。

### 5.4.3、文件清理策略

Kafka 中默认的日志保存时间为 7 天，可以通过调整如下参数修改保存时间。

* log.retention.hours，最低优先级小时，默认 7天。
* log.retention.minutes，分钟。
* log.retention.ms，最高优先级毫秒。
* log.retention.check.interval.ms，负责设置检查周期，默认 5 分钟。

那么日志一旦超过了设置的时间，怎么处理呢？

在Kafka 中提供的日志清理策略有 delete 和 compact两种。

#### 5.4.3.1、delete日志删除：将过期数据删除

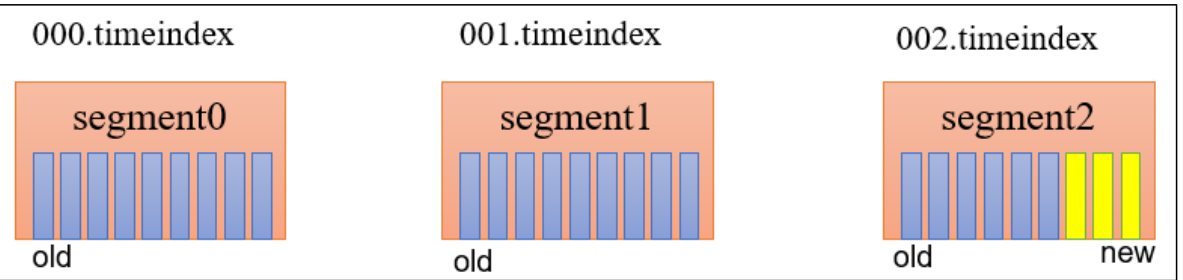
配置文件清理策略为日志删除，log.cleanup.policy = delete 所有数据启用删除策略。有删除的时候有两种方式，基于时间以及基于大小。

（1）基于时间：默认打开。以 segment 中所有记录中的最大时间戳作为该文件时间戳。

（2）基于大小：默认关闭。超过设置的所有日志总大小，删除最早的 segment。

log.retention.bytes，默认等于-1，表示无穷大。

如果一个 segment中有一部分数据过期，一部分没有过期，怎么处理？

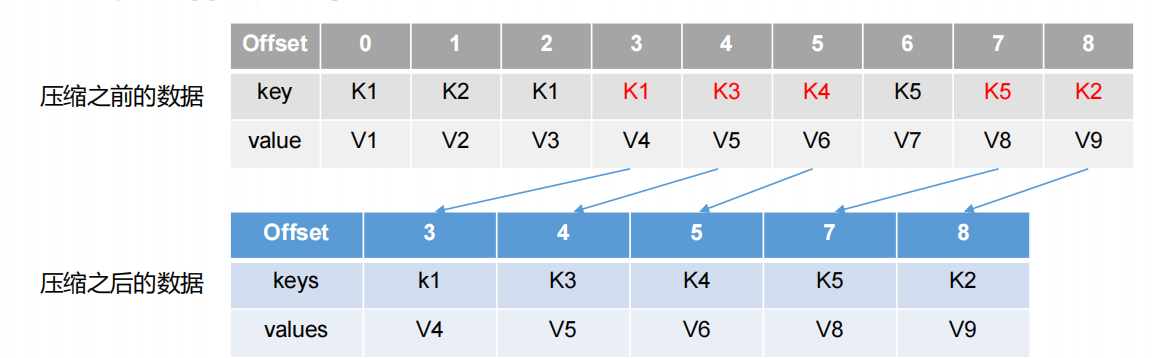


当segment文件中有一部分数据过期，一部分没有过期的时候，会以该文件中的最大时间戳作为过期时间，如果最大过期时间没有达到要清理的时间的话则不用清理，等待下次检查过期文件的来进行清理。

#### 5.4.3.2、compact日志压缩

compact日志压缩：对于相同key的不同value值，只保留最后一个版本。

log.cleanup.policy = compact 所有数据启用压缩策略



压缩后的offset可能是不连续的，比如上图中没有6，当从这些offset消费消息时，将会拿到比这个offset大 的offset对应的消息，实际上会拿到offset为7的消息，并从这个位置开始消费。

这种策略只适合特殊场景，比如消息的key是用户ID，value是用户的资料，通过这种压缩策略，整个消息集里就保存了所有用户最新的资料。

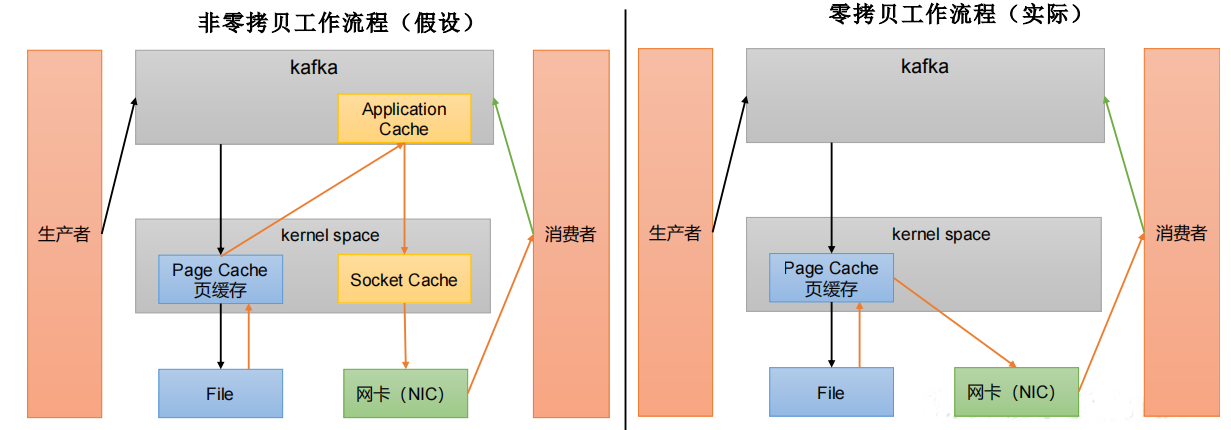
## 5.5、高效读写数据

* Kafka 本身是分布式集群，可以采用分区技术，并行度高
* 读数据采用稀疏索引，可以快速定位要消费的数据
* 顺序写磁盘，Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。
* 页缓存 + 零拷贝技术

### 5.5.1、页缓存 + 零拷贝技术

零拷贝：Kafka的数据加工处理操作交由Kafka生产者和Kafka消费者处理。Kafka Broker应用层不关心存储的数据，所以就不用走应用层，传输效率高。

PageCache页缓存：Kafka重度依赖底层操作系统提供的PageCache功能。当上层有写操作时，操作系统只是将数据写入PageCache。当读操作发生时，先从PageCache中查找，如果找不到，再去磁盘中读取。实际上PageCache是把尽可能多的空闲内存都当做了磁盘缓存来使用。



### 5.5.2、非零拷贝流程：

#### 5.5.2.1、概要

生产者生产消息，将消息发送到kafka集群，接着kafka会将消息内容交给linux内核进行处理，linux内核会将数据缓存起来一份，放到页缓存中，最后由linux内核将数据存储到磁盘中，当消费者消费消息的时候，kafka会先去页缓存中找，页缓存如果找不到的话，会去磁盘中读取，返回消息信息，同时也会在页缓存中存储一份数据，当kafka向消费者发送数据的时候（有可能kafka和消费者不在一台服务器上，需要跨节点通信，跨节点通讯需要网卡）， kafka会先将数据先发送到linux内核中的socket缓存中，再由socket缓存将数据通过网卡发送到消费者。

#### 5.5.2.2、非零拷贝消耗资源和速度慢的原因

在非零拷贝的过程中一共经历了4次拷贝数据的过程，从磁盘文件到linux内核页缓存，从内核页缓存到kafka，接着从kafka到内核的socket缓存，再从socket缓存到网卡。

从磁盘文件到linux内核，从socket缓存到网卡，是属于DMA拷贝（直接存储器访问），不经过cpu,消耗的资源和时间比较小

从内核页缓存到kafka，接着从kafka到内核的socket缓存，是属于cpu拷贝，而且需要在用户态和内核态之间来回切换，会消耗大量的资源和时间。

### 5.5.3、零拷贝流程

零拷贝其实是根据内核状态划分的，在这里没有经过CPU的拷贝，数据在用户态的状态下，经历了零次CPU拷贝，所以才叫做零拷贝，但不是说不拷贝。

#### 5.5.3.1、概要

生产者生产消息，将消息发送到kafka集群，接着kafka会将消息内容交给linux内核进行处理，linux内核会将数据缓存起来一份，放到页缓存中，最后由linux内核将数据存储到磁盘中，当消费者消费消息的时候，kafka会先去页缓存中找，页缓存如果找不到的话，会去磁盘中读取，返回消息信息，接着会直接通过网卡将数据发送到消费者，由消费者消费数据。因为kafka操作数据的过程是在生产者端和消费者端，broker应用层不操作数据，所以不需要将数据拷贝到broker应用层，可以直接由页面缓存通过网卡将数据发送到消费者。

#### 5.5.3.2、零拷贝速度快的原因

在零拷贝的过程中一共经历了2次拷贝数据的过程，从磁盘文件到linux内核页缓存，再从linux内核页缓存到网卡。

从磁盘文件到linux内核页缓存，再从linux内核页缓存到网卡，不经过cpu,消耗的资源和时间比较小，而且避免了需要在用户态和内核态之间来回切换，从而节省了大量的资源和时间。

# 6、Kafka 消费者

## 6.1、Kafka 消费方式

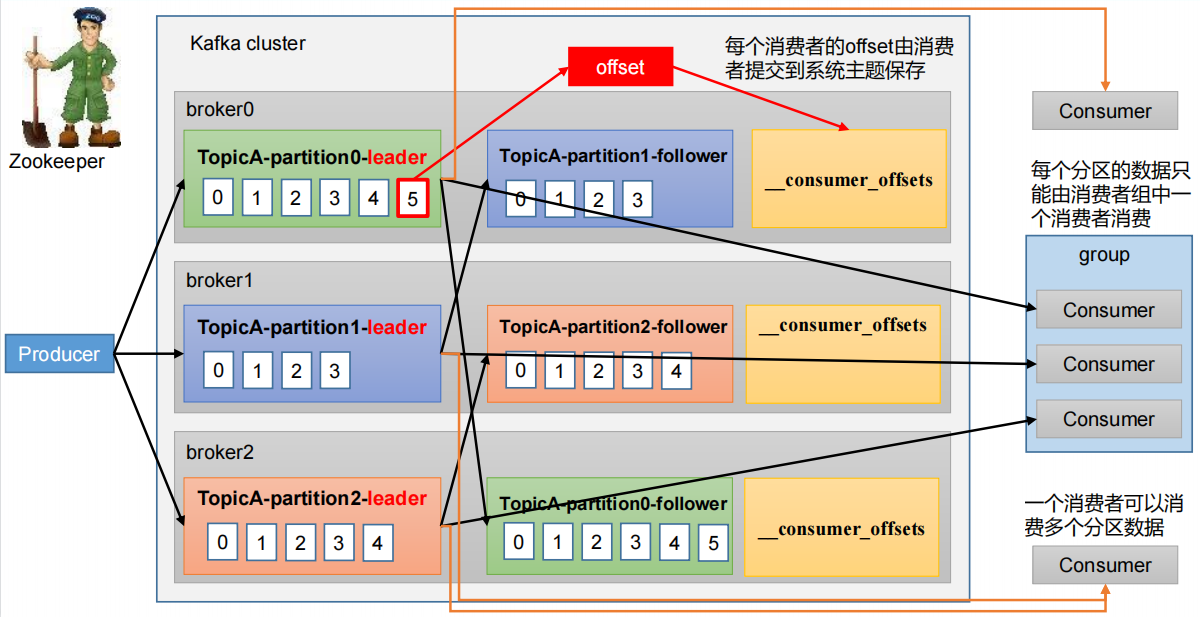
### 6.1.1、pull（拉）模式：

consumer采用从broker中主动拉取数据，Kafka采用这种方式。pull模式不足之处是，如果Kafka没有数据，消费者可能会陷入循环中，一直返回空数据。

### 6.1.2、push（推）模式：

由broker主动向消费者主动推送消息，Kafka没有采用这种方式的原因是，由broker决定消息发送速率，很难适应所有消费者的消费速率。

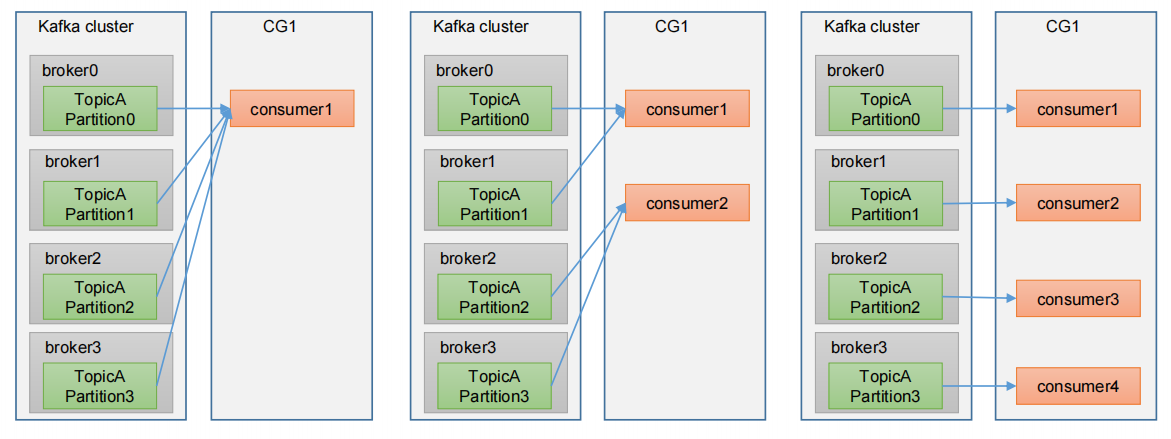
## 6.2、Kafka 消费者工作流程



生产者生产消息到达broker的各个分区中，消费者从分区的leader中读取消息，一个消费者可以消费多个分区数据，每个分区的数据只能由消费者组中一个消费者消费，每个消费者消费消息的offset，由消费者提交到\_\_consumer\_offsets系统主题保存。

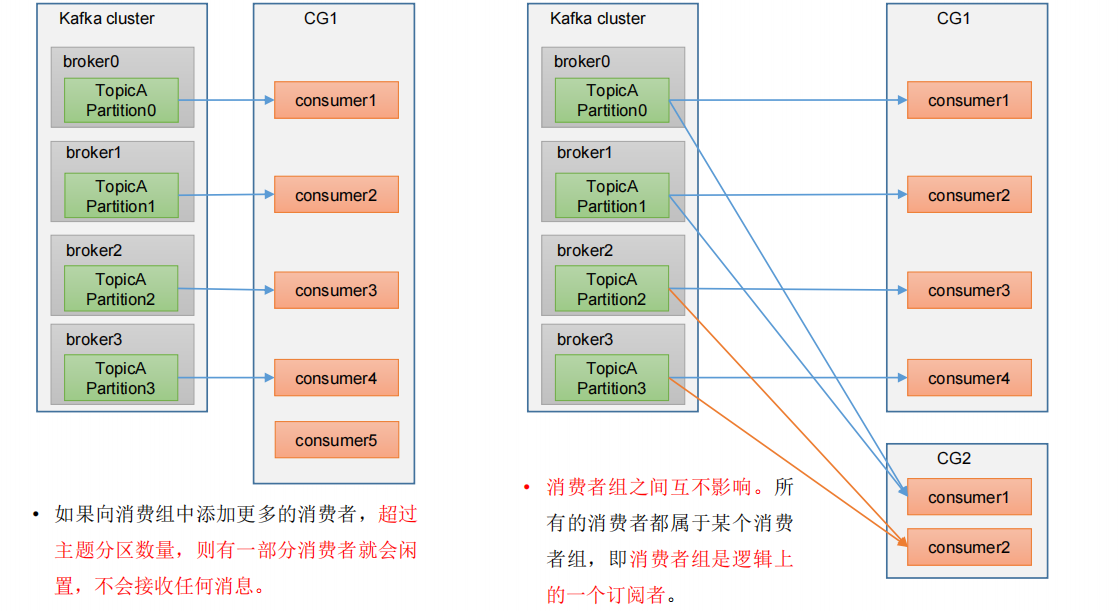
## 6.2、消费者组原理

### 6.2.1、概要

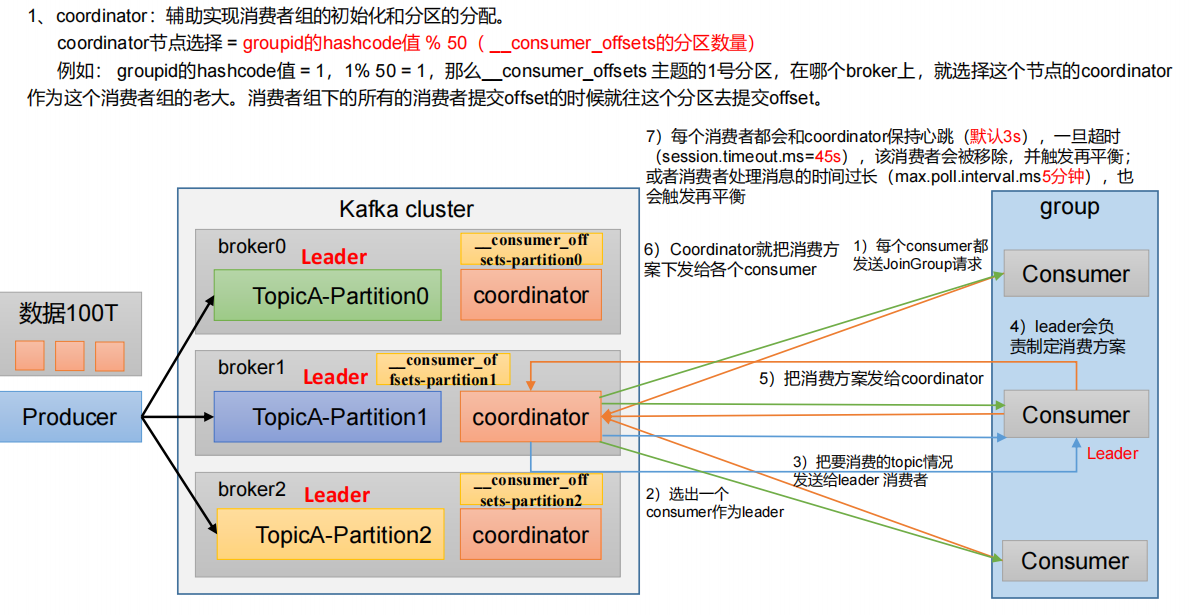


Consumer Group（CG）：消费者组，由多个consumer组成。形成一个消费者组的条件，是所有消费者的groupid相同。

* + 消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费。
  + 消费者组之间互不影响。所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

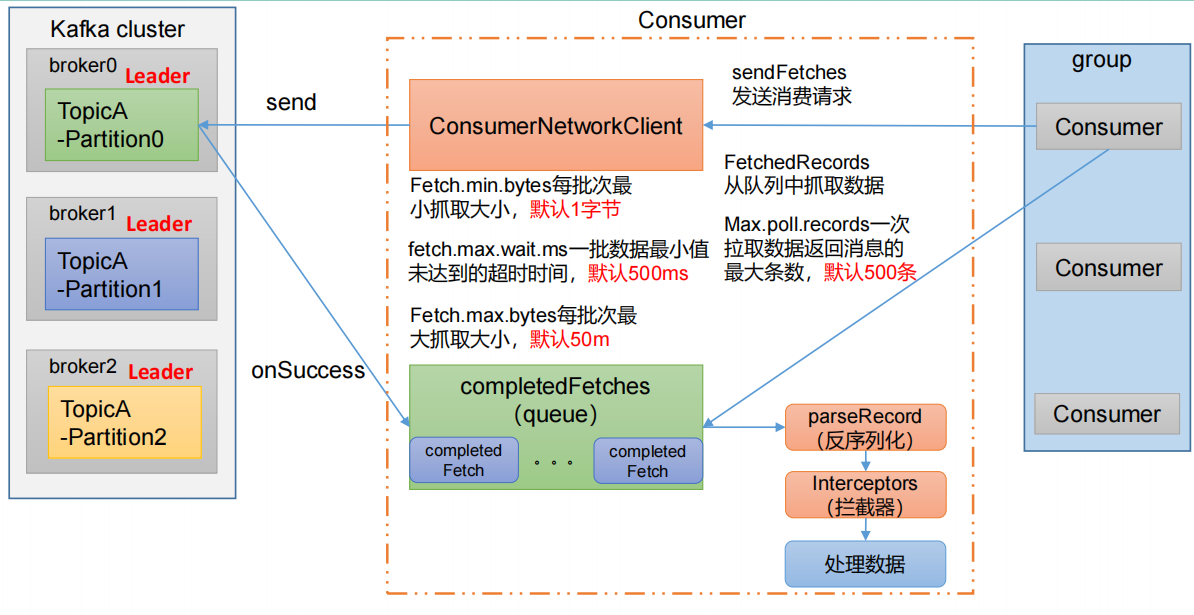


### 6.2.2、消费者组初始化流程



消费者组的初始化会借助coordinator辅助实现消费者组的初始化和分区的分配，首先每一个broker中都会存在一个coordinator，应该选择哪一个broker上的coordinator？coordinator的选择策略是根据groupId的hashcode值与offsets的分区数量求模(\_comsumer\_offsetsd的默认分区数是50，例如： groupid的hashcode值 = 1，1% 50 = 1，那么\_\_consumer\_offsets 主题的1号分区，在哪个broker上，就选择这个节点的coordinator作为这个消费者组的老大，消费者组下的所有的消费者提交offset的时候就往这个分区去提交offset)，选择出coordinator之后，这个消费者组的所有消费者都会向coordinator发送加入消费者组的请求，coordinator收到请求之后，会从所有的消费者中随机选择一个作为leader，然后将要消费的topic信息发送给leader，由leader负责制定消费方案，制定完相应的消费计划之后，再将这个消费计划发送给coordinator，然后coordinator将计划发给每一个消费者，每一个消费者按照制定的消费计划进行消费，在消费过程中每个消费者都会和coordinator保持心跳（默认3s），一旦超时（session.timeout.ms=45s），该消费者会被移除，并且原本该消费者的任务，也会被分配到其他的消费者上，触发再平衡；或者消费者处理消息的时间过长（max.poll.interval.ms5分钟），超过5分区未从分区中拉取数据，也会导致消费者被移除，并且原本该消费者的任务，也会被分配到其他的消费者上，触发再平衡。

### 6.2.3、消费者消费流程



消费者组要消费消息，首先会先创建消费者连接客户端（ConsumerNetworkClient），与kafka集群进行交互，调用sendFetches方法发送消费请求，进行抓取数据的初始化，抓取数据初始化主要有三个参数。

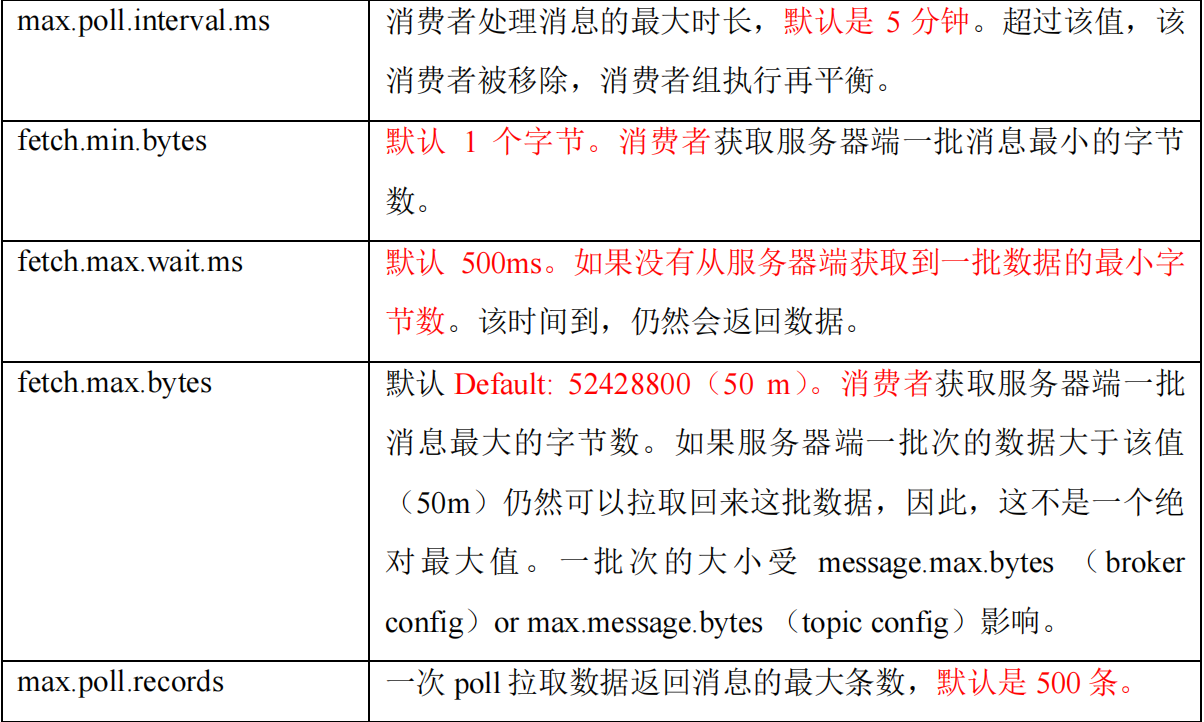
* Fetch.min.bytes每批次最小抓取大小（默认1字节）， 当一批次数据不满足最小的抓取大小（1字节），等待到达超时时间，也会将这些数据返回。
* fetch.max.wait.ms一批数据最小值未达到的超时时间（默认500ms），
* Fetch.max.bytes每批次最大抓取大小（默认50m），

抓取初始化准备完之后，调用send方法发送请求，通过请求成功的回调函数，将数据拉取回来，拉取回来的数据按照批次大小放到一个队列中（completedFetches），然后消费者会从队列（FetchedRecords）中抓取数据，Max.poll.records一次拉取数据返回消息的最大条数默认500条，然后经过反序列化、拦截器等进行数据处理，最终进行消费。

### 6.2.4、消费者重要参数

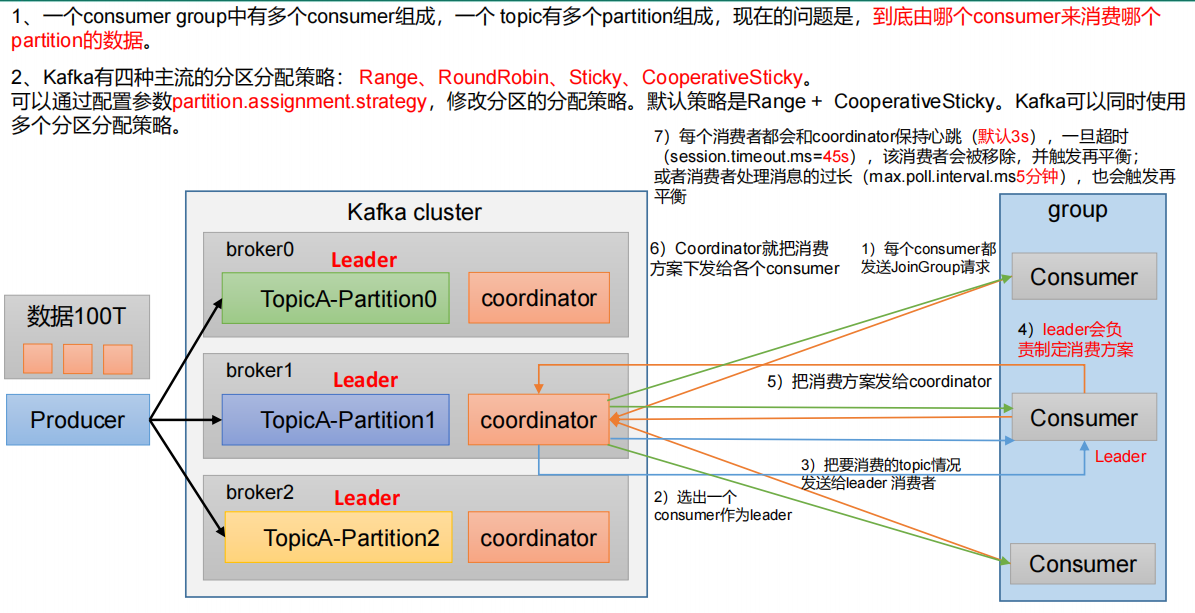






## 6.3、消费者 API

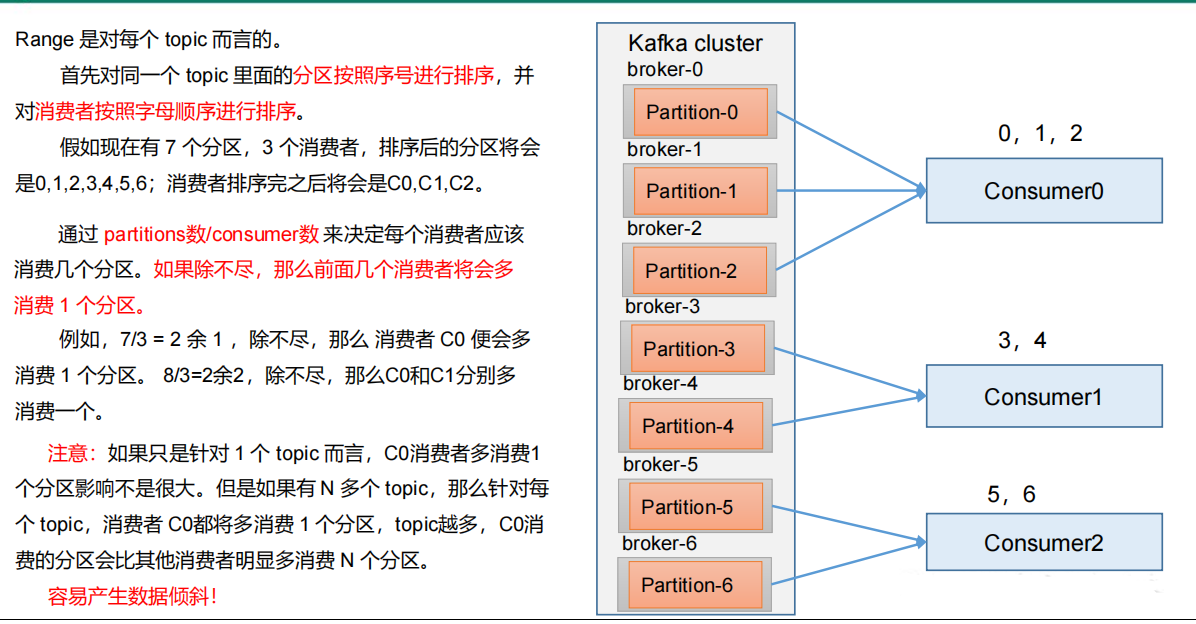
## 6.4、消费者组消费分区的分配策略以及再平衡



### 6.4.1、分配策略

Kafka有四种主流的分区分配策略：Range（范围）、RoundRobin（轮询）、Sticky(黏性)、CooperativeSticky(合作者黏性)。可以通过配置参数partition.assignment.strategy，修改分区的分配策略，默认策略是Range + CooperativeSticky，Kafka可以同时使用多个分区分配策略。

#### 6.4.1.1、Range 分区策略原理以及再平衡



Range 是对每个topic而言的，首先对同一个topic里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序，通过 partitions（分区数）/consumer（消费者个数）来决定每个消费者应该消费几个分区，如果除不尽，那么前面几个消费者将会多消费1个分区。消费者在进行消息消费的过程中，突然其中的一个消费者宕机，就会出现再平衡，消费者组会按照超时时间 45s 来判断它是否退出，时间到了45s 后，判断它真的退出就会把任务分配给其他的消费者进行消费，任务分配的方式是会整体被分配到其中的一个消费者上。

假如现在有7个分区，3个消费者，排序后的分区将会是0,1,2,3,4,5,6；消费者排序完之后将会是C0,C1,C2。7/3 = 2 余 1 ，除不尽，那么消费者 C0 便会多消费 1 个分区。 8/3=2余2，除不尽，那么C0和C1分别多消费一个分区。

0 号消费者：消费到 0、1、2号分区数据。

1 号消费者：消费到 3、4 号分区数据。

2 号消费者：消费到 5、6 号分区数据。

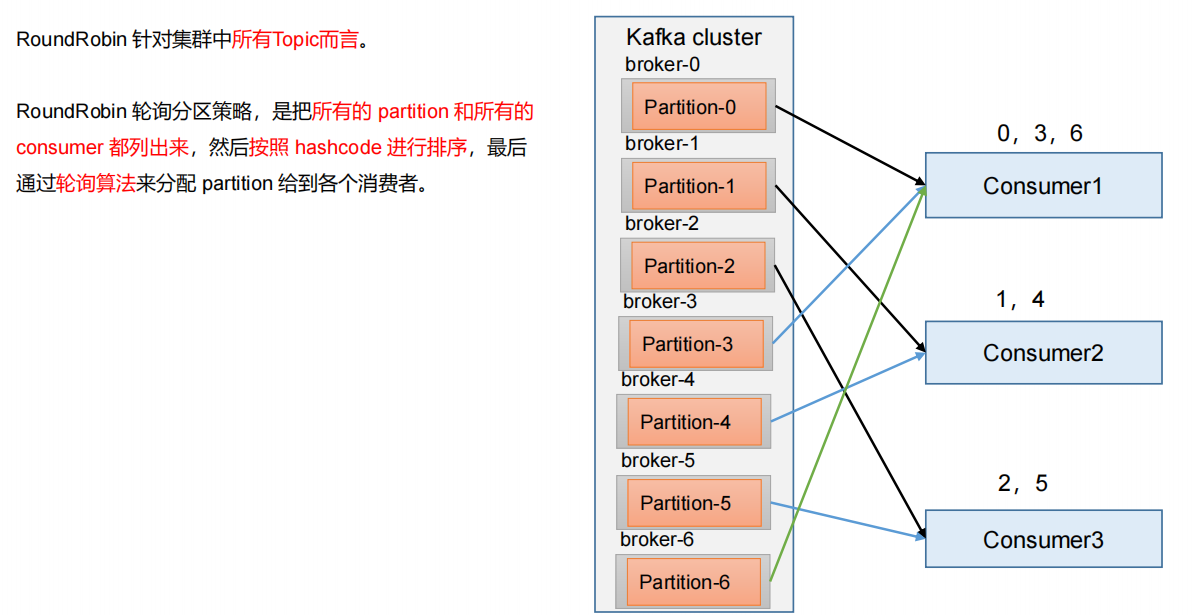
0 号消费者挂掉之后，0 号消费者的任务会整体被分配到 1 号消费者或者 2 号消费者。

1 号消费者：消费到 3、4、0、1、2号分区数据。

2 号消费者：消费到 5、6 号分区数据。

**注意**：如果只是针对 1 个 topic 而言，C0消费者多消费1个分区影响不是很大。但是如果有 N 多个 topic，那么针对每个 topic，消费者 C0都将多消费 1 个分区，topic越多，C0消费的分区会比其他消费者明显多消费 N 个分区。容易产生数据倾斜！

#### 6.4.1.2、RoundRobin 分区策略原理以及再平衡



RoundRobin 针对集群中所有Topic而言。RoundRobin 轮询分区策略，是把所有的 partition 和所有的consumer 都列出来，然后按照 hashcode 进行排序，最后通过轮询算法来分配 partition 给到各个消费者。消费者在进行消息消费的过程中，突然其中的一个消费者宕机，就会出现再平衡，消费者组会按照超时时间 45s 来判断它是否退出，时间到了45s 后，判断它真的退出就会把任务分配给其他的消费者进行消费，任务的分配方式是将已经宕机的消费者的分区数据按照轮询的方式添加到其他的消费者上。

0 号消费者：消费到 0、3、6 号分区数据

1 号消费者：消费到 1、4 号分区数据

2 号消费者：消费到 2、5 号分区数据

0 号消费者挂掉之后，0 号消费者的分区会轮询的方式分配到1号消费者或者 2 号消费者。

1 号消费者：消费到 1、4、0、6号分区数据

2 号消费者：消费到 2、5、3号分区数据

#### 6.4.1.3、Sticky 分区策略原理以及再平衡

首先会尽量均衡的放置分区到消费者上面，例如说7个分区，交给3个消费者进行消费，他的分配策略也是3,2,2的格式，和range类似，但不同的是黏性分区中的3不一定是第一个，而是随机的，尽可能的均衡，消费者在进行消息消费的过程中，突然其中的一个消费者宕机，就会出现再平衡，消费者组会按照超时时间 45s 来判断它是否退出，时间到了45s 后，判断它真的退出就会把任务分配给其他的消费者进行消费，任务的分配方式是会按照尽量保持均衡的方式进行分区。

0 号消费者：消费到 0、1 号分区数据。

1 号消费者：消费到 2、5、3 号分区数据。

2 号消费者：消费到 4、6 号分区数据。

0 号消费者挂掉之后，0 号消费者的分区以尽量保持均衡的方式分配到1号消费者或者 2 号消费者。

#### 6.4.1.4、Cooperative sticky

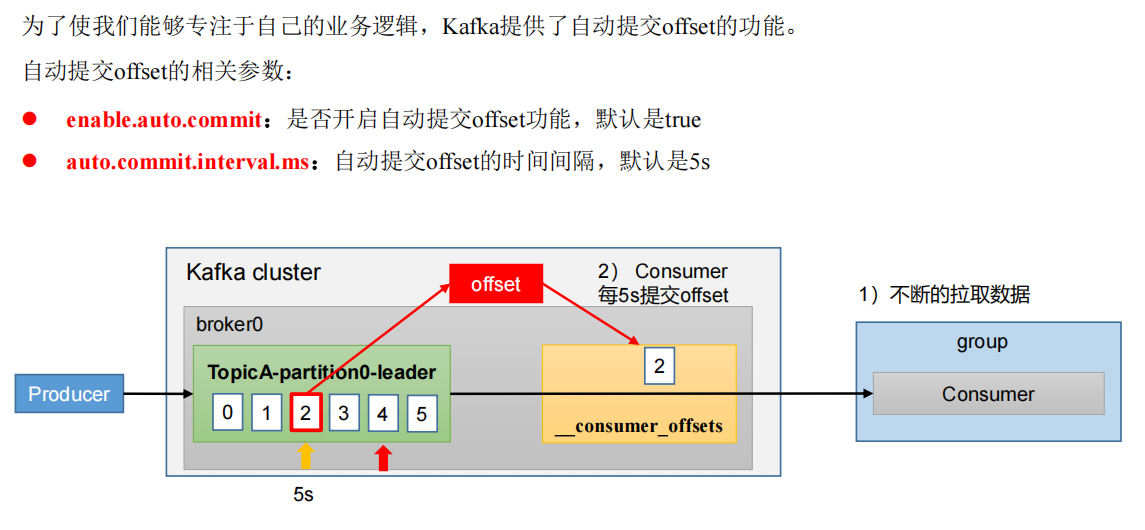
和sticky类似只是说支持了cooperative的rebalance。

## 6.5、offset 位移

### 6.5.1、概要

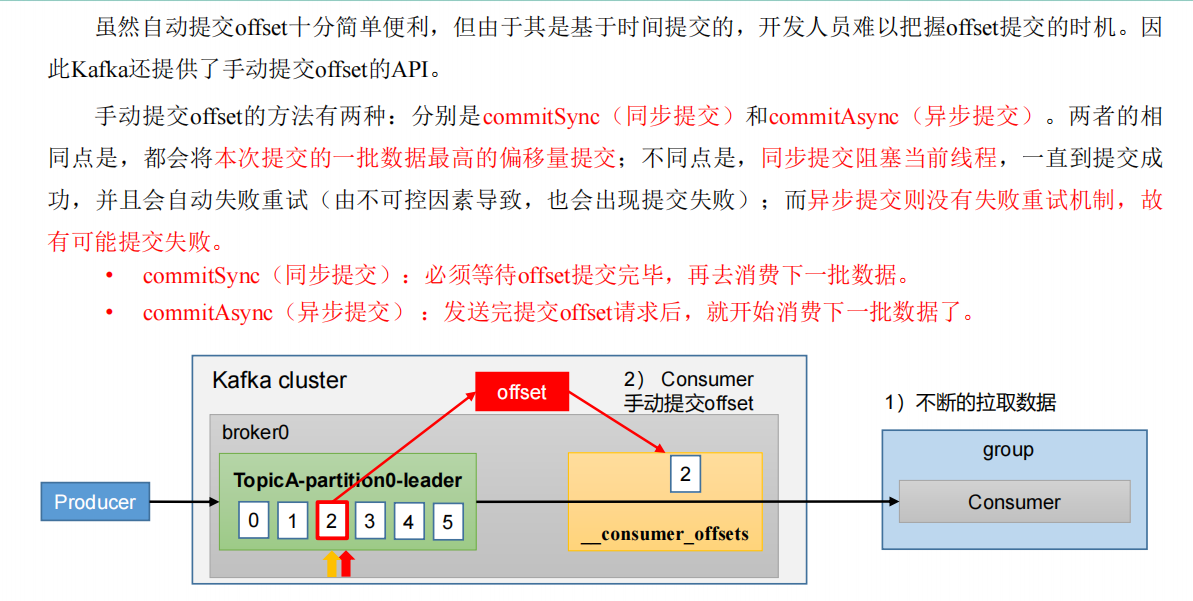
从0.9版本开始，consumer默认将offset保存在Kafka一个内置的topic中，该topic为\_\_consumer\_offsets ，\_\_consumer\_offsets 主题里面采用 key 和 value 的方式存储数据。key 是 group.id+topic+分区号，value 就是当前 offset 的值。每隔一段时间，kafka 内部会对这个 topic 进行compact，也就是每个 group.id+topic+分区号就保留最新数据。

### 6.5.2、自动提交 offset



生产者向分区中生产数据，消费者从分区中拉取数据，在拉取的过程中，消费者每间隔5s会提交消费的offset到系统主题中（\_consumer\_offsets）主题中。

### 6.5.3、手动提交 offset



自动提交offset十分简单便利，但由于其是基于时间提交的，开发人员难以把握offset提交的时机。因 此Kafka还提供了手动提交offset的API。

手动提交offset的方法有两种：分别是commitSync（同步提交）和commitAsync（异步提交）。

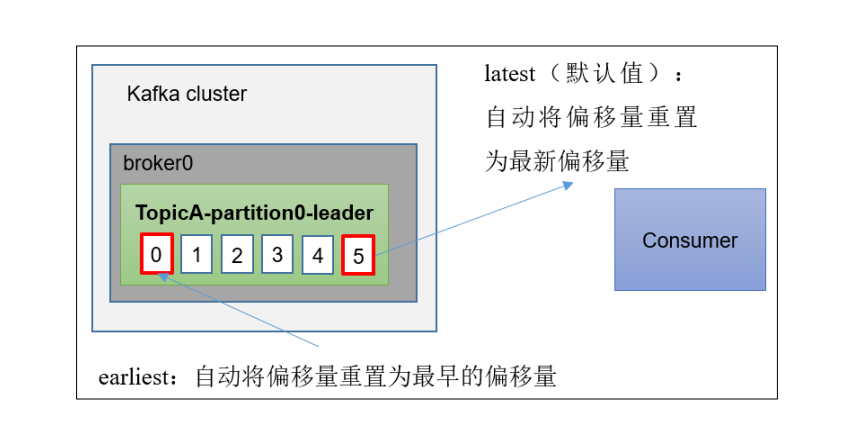
两者的相同点是都会将本次提交的一批数据最高的偏移量提交；

不同点是同步提交阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而异步提交则没有失败重试机制，故有可能提交失败。

* + commitSync（同步提交）：必须等待offset提交完毕，再去消费下一批数据。
  + commitAsync（异步提交） ：发送完提交offset请求后，就开始消费下一批数据了。

### 6.5.4、指定 Offset 消费

auto.offset.reset = earliest | latest | none 默认是 latest。

当 Kafka 中没有初始偏移量（消费者组第一次消费）或服务器上不再存在当前偏移量时（例如该数据已被删除），该怎么办？ 

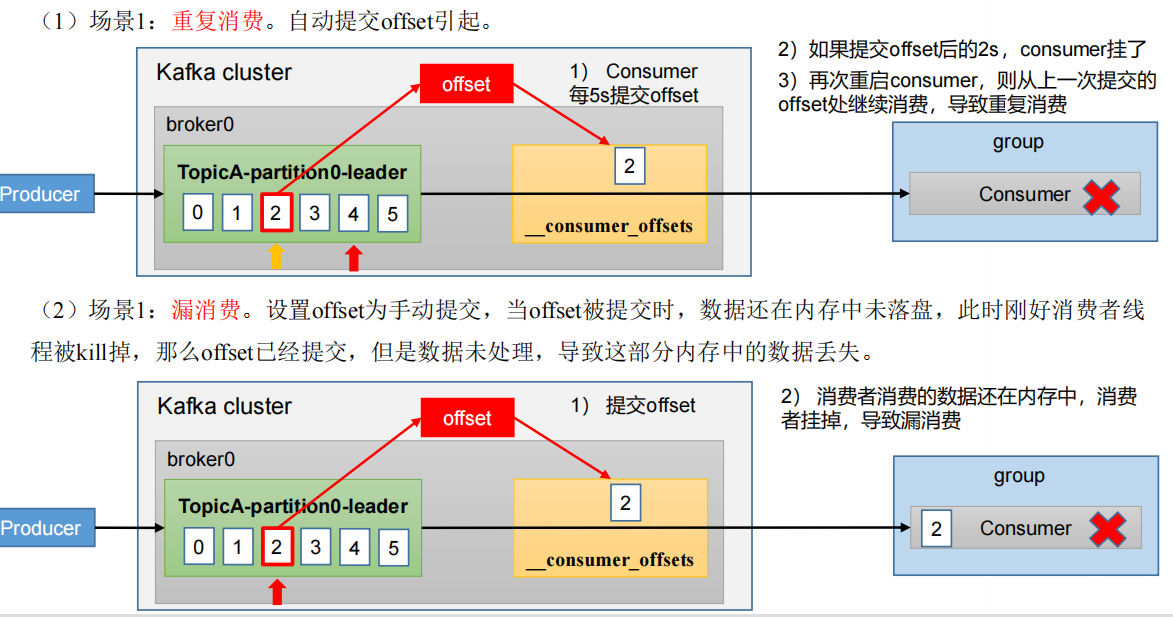
* earliest：自动将偏移量重置为最早的偏移量，--from-beginning。
* latest（默认值）：自动将偏移量重置为最新偏移量。
* none：如果未找到消费者组的先前偏移量，则向消费者抛出异常

### 6.5.5、指定时间消费

### 6.5.6、漏消费和重复消费

重复消费：已经消费了数据，但是 offset 没提交。

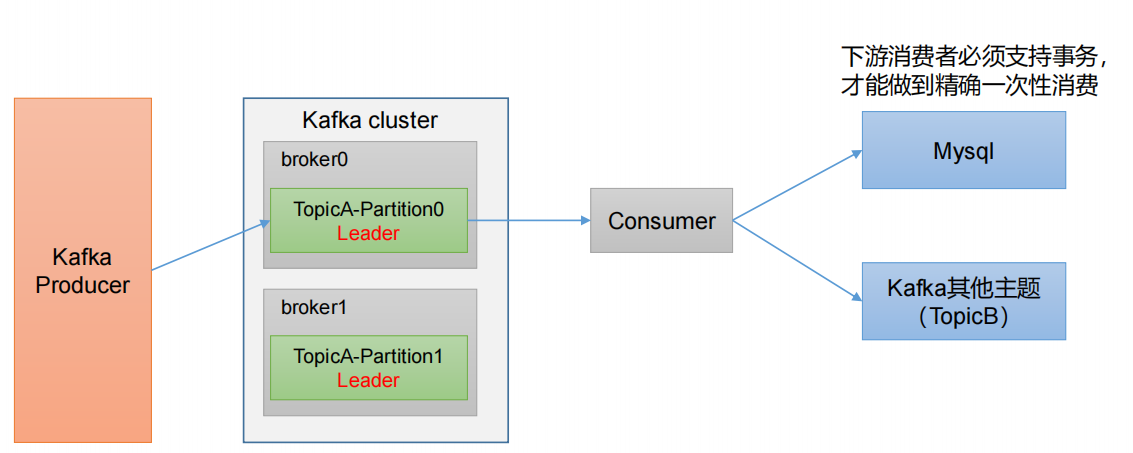
漏消费：先提交 offset 后消费，有可能会造成数据的漏消费。



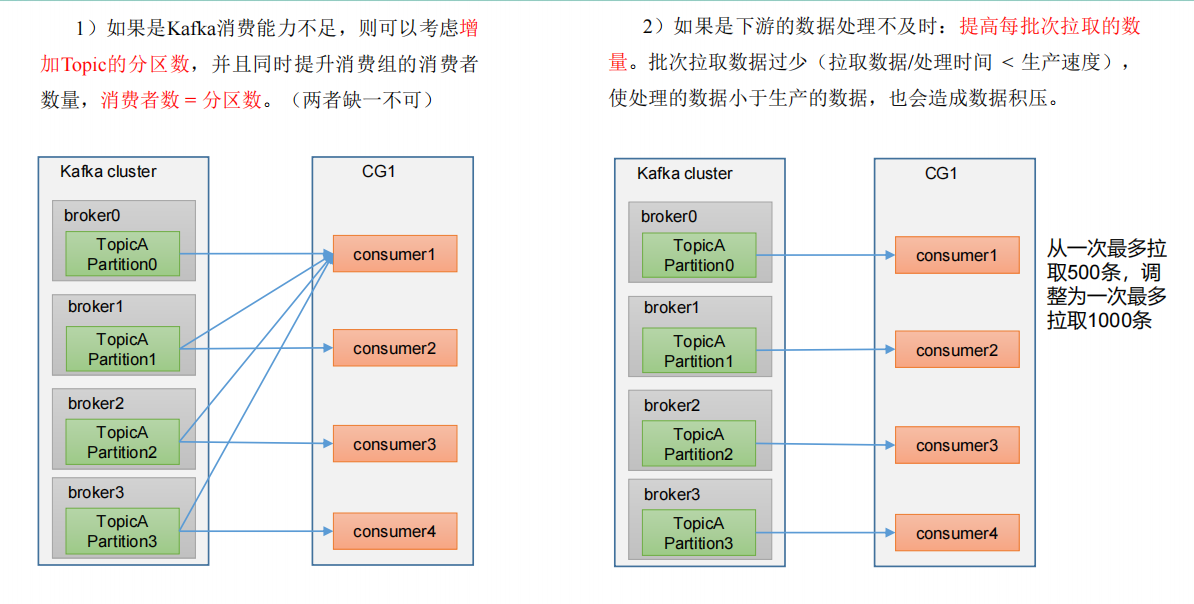
怎么能做到既不漏消费也不重复消费呢？使用消费者事务。

## 6.6、消费者事务

如果想完成Consumer端的精准一次性消费，那么需要Kafka消费端将消费过程和提交offset过程做原子绑定。此时我们需要将Kafka的offset保存到支持事务的自定义介质（比如MySQL）。



## 6.7、数据积压（消费者如何提高吞吐量）



# 7、Kafka-Eagle 监控

# 8、Kafka-Kraft 模式

## 8.1、Kafka-Kraft 架构

## 8.2、Kafka-Kraft 集群部署

## 8.3、Kafka-Kraft 集群启动停止脚本

# 9、集成 SpringBoot

# 10、调优-Kafka 硬件配置选择

## 10.1、场景说明

100 万日活，每人每天 100 条日志，每天总共的日志条数是 100 万 \* 100 条 = 1 亿条。 1 亿/24 小时/60 分/60 秒 = 1150 条/每秒钟。

每条日志大小：0.5k - 2k（取 1k）。

1150 条/每秒钟 \* 1k ≈ 1m/s 。

高峰期每秒钟：1150 条 \* 20 倍 = 23000 条。

每秒多少数据量：20MB/s。

## 10.2、服务器台数选择

服务器台数= 2 \* （生产者峰值生产速率 \* 副本 / 100） + 1

= 2 \* （20m/s \* 2 / 100） + 1

= 3 台

建议 3 台服务器。

## 10.3、磁盘选择

kafka 底层主要是顺序写，固态硬盘和机械硬盘的顺序写速度差不多。

建议选择普通的机械硬盘。

每天总数据量：1 亿条 \* 1k ≈ 100g

100g \* 副本 2 \* 保存时间 3 天 / 0.7 ≈ 1T

建议三台服务器硬盘总大小，大于等于 1T。

## 10.4、内存选择

Kafka 内存组成：堆内存 + 页缓存

1. Kafka 堆内存建议每个节点：10g ~ 15g

在 kafka-server-start.sh 中修改

if [ "x$KAFKA\_HEAP\_OPTS" = "x" ]; then

export KAFKA\_HEAP\_OPTS="-Xmx10G -Xms10G"

fi

2）页缓存：页缓存是 Linux 系统服务器的内存。我们只需要保证 1 个 segment（1g）中25%的数据在内存中就好。

每个节点页缓存大小 =（分区数 \* 1g \* 25%）/ 节点数。例如 10 个分区，页缓存大小=（10 \* 1g \* 25%）/ 3 ≈ 1g

建议服务器内存大于等于 11G。

## 10.5、CPU 选择

num.io.threads = 8 负责写磁盘的线程数，整个参数值要占总核数的 50%。 num.replica.fetchers = 1 副本拉取线程数，这个参数占总核数的 50%的 1/3。

num.network.threads = 3 数据传输线程数，这个参数占总核数的 50%的 2/3。

建议 32 个 cpu core。

## 10.6、网络选择

网络带宽 = 峰值吞吐量 ≈ 20MB/s 选择千兆网卡即可。

100Mbps 单位是 bit；10M/s 单位是 byte ; 1byte = 8bit，100Mbps/8 = 12.5M/s。

一般百兆的网卡（100Mbps ）、千兆的网卡（1000Mbps）、万兆的网卡（10000Mbps）。

# 11、调优- Kafka 生产者

## 11.1、Kafka 生产者核心参数配置

## 11.2、生产者如何提高吞吐量

## 11.3、数据可靠性

## 11.4、数据去重

## 11.5、数据有序

## 11.6、数据乱序

# 12、调优-Kafka Broker

## 12.1、Broker 核心参数配置

## 12.2、服役新节点/退役旧节点

## 12.3、增加分区

## 12.4、增加副本因子

## 12.5、手动调整分区副本存储

## 12.6、Leader Partition 负载平衡

## 12.7、自动创建主题

## 12.8、Leader Partition 负载平衡

# 13、调优-Kafka 消费者

## 13.1、Kafka 消费者核心参数配置

## 13.2、消费者再平衡

## 13.3、指定 Offset 消费

## 13.4、指定时间消费

## 13.5、消费者事务

## 13.6、消费者如何提高吞吐量

# 14、调优-Kafka 总体

## 14.1、如何提升吞吐量

## 14.2、数据精准一次

## 14.3、合理设置分区数

## 14.4、单条日志大于 1m

## 14.5、服务器挂了

## 14.6、集群压力测试

# 15、源码-Kafka 生产者源码

# 16、源码-Kafka 消费者源码

# 17、源码-Kafka 服务器源码

# 18、面试题